Efficient Hardware Implementations of High Throughput SHA-3 Candidates Keccak, Luffa and Blue Midnight Wish for Single- and Multi-Message Hashing

Abdulkadir Akın, Aydın Aysu, Onur Can Ulusel, and Erkay Savaş

Sabanci University, Faculty of Engineering and Natural Sciences, Orhanli, 34956 Tuzla, Istanbul, Turkey
{abdulkadir, aydinaysu, onuracansel}@su.sabanciuniv.edu, erkays@sabanciuniv.edu

Abstract. In November 2007 NIST announced that it would organize the SHA-3 competition to select a new cryptographic hash function family by 2012. In the selection process, hardware performances of the candidates will play an important role. Our analysis of previously proposed hardware implementations shows that three SHA-3 candidate algorithms can provide superior performance in hardware: Keccak, Luffa and Blue Midnight Wish (BMW). In this paper, we provide efficient and fast hardware implementations of these three algorithms. Considering both single- and multi-message hashing applications with an emphasis on both speed and efficiency, our work presents more comprehensive analysis of their hardware performances by providing different performance figures for different target devices. To our best knowledge, this is the first work that provides a comparative analysis of SHA-3 candidates in multi-message applications. We discover that BMW algorithm can provide much higher throughput than previously reported if used in multi-message hashing. We also show that better utilization of resources can increase speed via different configurations. We implement our designs using Verilog HDL, and map to both ASIC and FPGA devices (Spartan3, Virtex2, and Virtex 4) to give a better comparison with those in the literature. We report total area, maximum frequency, maximum throughput and throughput/area of the designs for all target devices. Given that the selection process for SHA3 is still open; our results will be instrumental to evaluate the hardware performance of the candidates.

Keywords: cryptographic hash functions, hardware implementation, SHA-3

1. Introduction

Cryptographic hash functions reduce arbitrary length input messages to a digest of fixed length. The need for cryptographic hash functions was first identified by Diffie and Hellman for digital signature scheme [1]. During 1970s, many researchers helped providing (e.g. Rabin [2] and Merkle [3]) the definitions, requirements and constructions for cryptographic hash functions. Easy computation, non-invertibility, strong/weak collision resistance and ciphertext indistinguishability are the main properties of secure cryptographic hash functions.

The need for efficient and secure hash functions was well understood during the 1980s. In order to meet this demand, SHA-1 and SHA-2 hash functions were published by the National Institute of Standards and Technology (NIST) in 1993 and 2002, respectively. Powerful attacks on SHA-1 [13] and similarly constructed SHA-2 variants [14], led to the initiation of SHA-3 open competition by NIST [4]. 51 candidates of the SHA-3 competition passed round 1, and 14 candidates advanced to the round 2 [4]. The final round candidates are scheduled for 2010 and the winner will be announced in 2012.

Hardware implementations of cryptographic algorithms are much more secure than software realizations [5]. In addition, they can be optimized to satisfy application's specific requirements, e.g. higher performance, low area, low power, better resource utilization. Since different aspects of hardware performances of the SHA-3 candidates play a significant role in the selection process, comparisons of hardware implementation of 14 remaining candidates are given in detail in [6] and [7]. The hardware architectures [6][7] are designed to provide the highest throughput for single message hashing (SMH), whereby hardware is assumed to process a single message stream at a time. It has also been shown in [8] that pipelined architectures increase the performance of Luffa for multi-message hashing (MMH), where the hardware processes more than one message concurrently.

Paucity of hardware implementations of SHA-3 candidates that exploit MMH to achieve higher performance calls for a more thorough study of the issue. To reveal the true potentials of SHA-3 candidates, we developed two hardware architectures each for the three high throughput SHA-3 candidates (six in total), namely Keccak, Luffa, and Blue Midnight Wish (BMW)[1]; one for SMH and the other for MMH applications. Our implementation results show that different configurations and design techniques result in higher performance for a given candidate. This study also aims to reveal as many aspects (e.g., frequency, throughput, latency, area efficiency, target device) pertaining to hardware performances of the three candidates as possible to enhance the insight on the algorithms for the selection process.

We present synthesis results of six hardware architectures for four target devices: Spartan 3, Virtex II and Virtex IV FPGAs, and 90nm ASIC. 256-bit versions of each algorithm are implemented and all architectures are verified using the reference software/hardware submissions of the competitors. We mainly target high throughput hardware architectures of the three candidates, *without unnecessarily increasing the area*. The area/performance results of our SMH and MMH implementations are compared with the best-performance architectures proposed in [6][7][8][11][12]. The implementation results show that our architectures provide better efficiency in majority of the cases.

The rest of the paper is organized as follows. Section 2 explains the reason for selecting Keccak, Luffa and BMW for hardware implementation and Section 3 explains MMH methodology for hardware design. Overview of Keccak, Luffa and BMW are presented in Section 4. The proposed hardware architectures for Keccak, Luffa and BMW will be described in Section 5. The implementation results will be given in Section 6. Section 7 will conclude the paper.

2. Selection Process of Three SHA-3 Candidates

In [7], the hardware implementation results of 14 second-round candidate SHA-3 algorithms are presented. All the implementations are synthesized using a uniform tool chain, standard-cell library, target technology, and optimization heuristics. The implementation results of [7] are summarized in the second column of Table 1. As seen from Table 1, Keccak gives the highest throughput of 21.23 Gbit/s while Luffa comes second with the throughput of 13.74 Gbit/s. The design methodology in [7] targets high throughput architectures for SMH applications and does not inspect architectures that are tailored for MMH applications. In general, the SMH design methodology favors single pipelining for each round. In the case of MMH, the hardware can be fully pipelined and pipeline stages can be fully utilized. In order to gain an insight on true potentials of the SHA-3 candidates in hardware implementation with the MMH applications in mind, we normalize the throughput results according to area and frequency values given in [7], as shown in the last three columns of Table 1.

Table 1. Normalized throughput values for the SHA-3 candidates

Algorithm	Throughput (Gbit/s)	Throughput Normalized to 100 MHz	Throughput Normalized to 100 GE	Throughput Normalized to 100 MHz and 100 GE
BLAKE	3.97	2.33	8.70	5.10
BMW	5.36	51.22	3.16	30.18
CubeHash	4.67	3.20	7.92	5.44
ECHO	2.25	1.58	1.59	1.12
Fugue	4.09	1.60	8.85	3.46
Grøstl	6.29	2.33	10.77	3.99
Hamsi	5.57	3.20	9.49	5.46
JH	4.99	1.31	8.49	2.23
Keccak	21.23	4.35	37.70	7.73
Luffa	13.74	2.84	30.56	6.33
Shabal	3.28	1.02	5.98	1.87
SHAvite	3.15	1.38	5.49	2.41
SIMD	0.92	1.42	0.89	1.37
Skein	2.50	5.12	2.45	5.02

The normalization suggests some interesting results; e.g. BMW having high performance. We must note that the values obtained through normalization *are in abstract level and not used as an objective in our design process*– and our implementations confirm that they are not achieved –; but they provide a deeper understanding and intuition to the multi-variable (e.g. area, clock frequency, and throughput) optimization problem for throughput comparison in MMH. Normalization results show that Keccak and Luffa have the highest throughputs (37.7 Gbit/s and 30.6 Gbit/s) under area normalization as well as non-normalized results (21.23 Gbit/s and 13.74 Gbit/s) since they are relatively light-weight designs while BMW is quite heavy. But, surprisingly, BMW achieves the highest throughput under

[1] We explain the selection technique for these three candidates in Section 2.

frequency (51.22 Gbit/s) and frequency/area (31.18 Gbit/s) normalizations. For the frequency/area normalization, Keccak and Luffa follows BMW with throughputs of 7.73 Gbit/s and 6.33 Gbit/s, respectively.

Since hardware performance is one of the most important factors for the SHA-3 hash function; Keccak, Luffa and BMW are implemented in this work using the multi-message design methodology which utilizes pipelined architectures and re-timing techniques. While high throughput is always the primary goal, our architectures are designed to achieve this goal with minimum area for each design.

3. Multi-Message Hashing Methodology for Hardware

In literature, Eq. 1 is generally used for calculating throughput for SMH architectures.

$$Throughput = \frac{Blocksize}{latency} \times Frequency \qquad (1)$$

where *latency* is the number of clock cycles required for processing one message block and *Blocksize* is the length of the message block which is processed by the hash function at a given time.

The formula for calculating throughput of the MMH is given in Eq 2.

$$Throughput = \frac{Blocksize \times \#MH}{latency} \times Frequency \qquad (2)$$

#MH is the number of messages that can be simultaneously hashed at a given time, which is equal to the number of pipeline stages in a single round of a hash algorithm. If the algorithm is not round-based (e.g., BMW) then *#MH* is equal to the number of total pipeline stages. For MMH applications, many messages can be simultaneously processed due to pipelining. Pipelining increases both the frequency and number of messages that can be hashed in parallel, which has a positive effect on throughput. However, if the number of messages that will be hashed is less than the number of the pipeline stages, full utilization of the architecture cannot be achieved. As a result, the throughput of the MMH algorithm decreases due to the latency overhead. Therefore in order to show the full potential of the MMH architectures, number of the messages that will be hashed concurrently is assumed to be more than or equal to the number of pipeline stages.

In pipelined datapath, pipeline registers are used to relay data from one stage to the next. In addition, certain input signals need to be forwarded to the subsequent pipelined stages through so-called synchronization registers. The need for pipeline and synchronization registers will, however, have an adverse effect on area, resulting in a poor area/frequency tradeoff.

Hardware replication in both FPGA and ASIC designs (e.g. having two identical circuits to compute the hash values of two independent messages) can also increase the throughput of the hardware for MMH applications. However, increasing the frequency by using efficient pipelining may yield better area/throughput tradeoff than hardware replication for MMH. Configurations, where hardware replication and efficient pipelining are used together, provide further increase in throughput.

4. Brief Description of Selected Algorithms

4.1 Keccak Algorithm

Keccak algorithm [4] uses *KECCAK-f* permutation which consists of a number of simple rounds with logical operations and bit permutations. Each round has 5 steps and 24 rounds form a *KECCAK-f* permutation. The input and output of a Keccak round are 5×5 matrices whose entries are 64-bit words. The round formulae are given in Fig. 1.

In Fig. 1, A and RC (round constant) are inputs, $\oplus$ (*XOR*), *AND*, and *NOT* represent bitwise logical operations. The output of a round is formed on the 5×5 matrix A. The variables x and y represent the matrix index and the operations on x and y are done modulo 5. $ROT(I[x, y], j)$ denotes the cyclic shift operation of the 64-bit number at $I[x, y]$ by the amount of j to the left. In ρ and π steps, the rotation is done by the amount of $r[x, y]$. In each round, a different RC is used. The round constants are given in the specifications of the candidate [4].

```
Round[b](A, RC)
  θ STEP
    $C[x] = A[x, 0] \oplus A[x, 1] \oplus A[x, 2] \oplus A[x, 3] \oplus A[x, 4]$    $\forall x$ in 0…4
    $D[x] = C[x-1] \oplus ROT(C[x+1], 1)$    $\forall x$ in 0…4
    $A[x, y] = A[x, y] \oplus D[x]$    $\forall(x, y)$ in (0…4,0…4)
  ρ AND π STEPS
    $B[y, 2x+3y] = ROT(A[x, y], r[x, y])$    $\forall(x, y)$ in (0…4,0…4)
  χ STEP
    $A[x, y] = B[x, y] \oplus ((NOT\ B[x+1, y])\ AND\ B[x+2, y])$    $\forall(x, y)$ in (0…4,0…4)
  ι STEP
    $A[0, 0] = A[0, 0] \oplus RC$
```

Fig. 1 Round operations of a *KECCAK-f* permutation

```
Keccak[r, c, d](M)
  Initialization and padding
    $S[x, y] = 0$    $\forall(x, y)$ in (0…4, 0…4)
    $P = M || \text{byte}(d) || \text{byte}(r/8) || \text{0x01} || \text{0x00} || ... || \text{0x00}$
  Absorbing Phase for every block $P_i$ in P
    $S[x, y] = S[x, y] \oplus P_i[x + 5y]$    $\forall(x, y)$ such that $x+5y < r/w$
    $S = KECCAK\text{–}f[r + c](S)$
  Squeezing Phase
  Z = empty string
  while output is requested
    $Z = Z || S[x, y]$    $\forall(x, y)$ such that $x+5y < r/w$
    $S = KECCAK\text{–}f[r+c](S)$
  return Z
```

Fig. 2 Phases of Keccak algorithm

Keccak has an absorbing and a squeezing phase. The input message is divided into blocks of 1088-bit and *XOR*ed onto a part of the state (which is initially zero and 1600-bit long) and the result is passed through a *KECCAK-f* permutation. The output is truncated to 256 bits. The phases are detailed in Fig 2, where Z is the output, and r = 1088, c = 512, w = 64, and d = 32 in our implementations. They are the values used in the implementation in [7].

4.2 Luffa Algorithm

Luffa algorithm [4] employs a variant of sponge function [9][10]. It utilizes s-boxes, and *XOR* and shift operations to hash a message. The input block sizes can be 224, 256, 384 or 512 bits, processed as 32-bit data words. Luffa's compression function is known as the round function, which comprises one Message Injection (*MI*) and one Permutation (*P*) stage.

MI module combines the hash values of previous message blocks (i.e., H_0^{i-1}, H_1^{i-1}, H_2^{i-1}), with the current message block (M^i). A message block in Luffa-256 can be represented by the matrix over a ring $GF(2^8)^{32}$. Block diagram of *MI* for Luffa-256 can be seen in Fig. 3. The inputs are the current message block and the hash values calculated by the three permutation blocks for the previous message block, each of which is 256 bits and have a constant initial value. The symbol ⊕ in Fig 3 represents a three input *XOR* operation and ⊗ represents a single multiplication in $GF(2^8)^{32}$ by constant 2.

The permutation stage (*P*) for Luffa-256 is made of three permutation blocks, which work in parallel and each block receives one of the 256-bit outputs of the *MI* as input. These blocks are referred as *Permute* blocks. Each *Permute* block starts with a permutation of the input which is called *tweak*, and then iterates 8 rounds of the *Step* function shown in Fig. 4.

Each *Step* function in three Permute blocks processes data in 32-bit words, denoted as a_k, $0 \leq k < 8$ in Fig. 4. There are three submodules in *Step* function referred as *SubCrumb*, *MixWord*, and *AddConstant*. The *SubCrumb* module is a nonlinear permutation implemented by 32 identical s-boxes (4-bit input, 4-bit output). The s-box can be shown as a mapping defined as $s[16] = \{7, 13, 11, 10, 12, 4, 8, 3, 5, 15, 6, 0, 9, 1, 2, 14\}$.

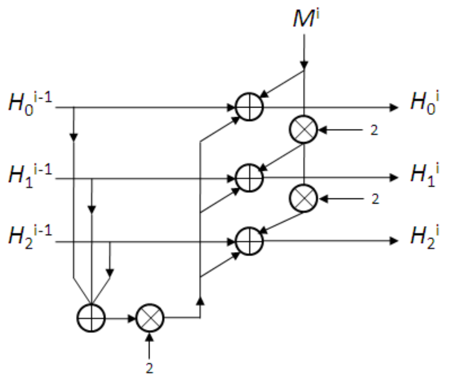
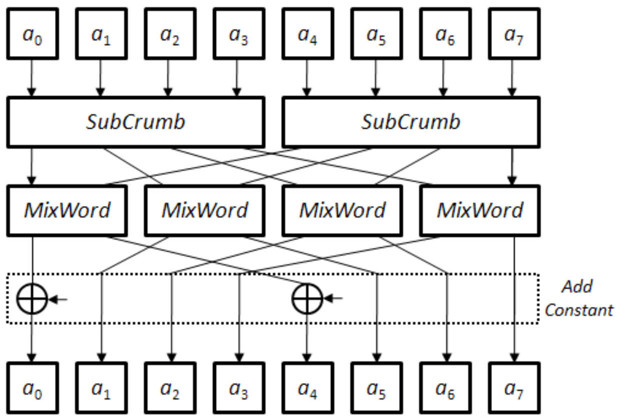


Fig. 3 Block diagram of the Message Injection module



a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
SubCrumb
SubCrumb
MixWord
MixWord
MixWord
MixWord
Add Constant
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7

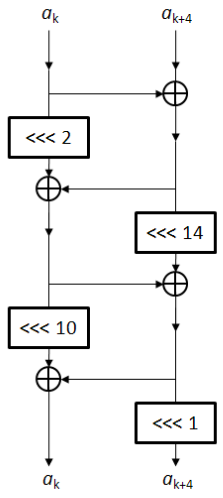


Fig. 4 Block diagram of the *Step* module

Fig. 5 Block diagram of *MixWord* function

MixWord is a Feistel ladder of 4 rounds, which is used to mix two words together. The block diagram of the *MixWord* is shown in Fig. 5. *AddConstant* module performs two *XOR* operations on the words a_0 and a_4 with predetermined constants. These constants differ for each round of the step function; which can be hardwired in the implementation.

The resulting hash values of the *Step* modules H_0^i, H_1^i, H_2^i, are given as inputs to the next message block. Once the last block of the message is computed, a blank round using a 256-bit all-zero message is computed and the output hash of the message is found by *XOR*ing the final results of the three *Step* functions.

4.3 Blue Midnight Wish Algorithm

The BMW hash function [4] uses quadrupled pipe $\{Q_a, Q_b\}$ (each of which is m-bit variable) and double pipe H (which is an m-bit variable) for iteratively computing new Q_a, Q_b, H values and the message digest, where m is the message block size. In generic description, BMW uses three steps: *preprocessing*, *hash computation* and *finalization*. *Preprocessing* step involves padding, parsing and initialization of variables as many hash algorithms use. The block diagram of the *hash computation* and *finalization* steps of the BMW algorithm are shown in Fig. 6. Hash computation and finalization steps involve three functions f_0, f_1 and f_2.

The function f_0 is used to compute the first part of quadrupled pipe (Q_a) by diffusing the message block M and double pipe H, where H is initialized to a constant value.

The function f_1 is used with two sub-functions *expand1* and *expand2*. The function f_1 takes M and Q_a as inputs and using a technique called "multi-permutation" generates Q_b as an output. Its designers propose that the security of BMW can be increased with increasing the *expand1* rounds and decreasing the *expand2* rounds. However, *expand1* is more complex than *expand2,* therefore designers recommend using two rounds for *expand1* and fourteen rounds for *expand2*.

The function f_2 is used in folding (compression) part of the BMW algorithm and it reduces $3m$-bit of M, Q_a, and Q_b to m-bit new double pipe H.

The basic operations of the BMW algorithm are addition and subtraction modulo 2^{32}, shift, rotate, and bitwise XOR.

Finalization step is similar to the *hash computation*; the only difference is that final step uses constant value instead of message block to form m-bit H^{final}. The least significant n-bit (length of resulting hash value) of H^{final} are given as hash of the message.

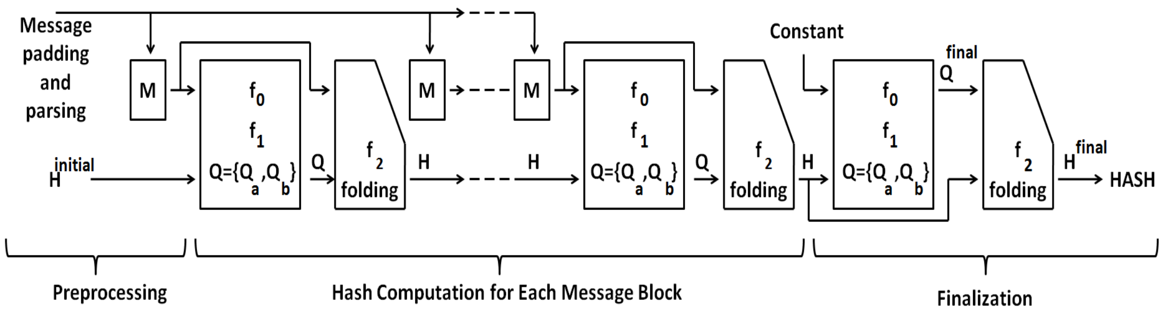


Fig. 6 Representation of BMW algorithm

5. Hardware Implementations

Pipelined hardware architectures, generally, are not suitable for SMH applications since only one stage would be active at any instance, resulting in a very low utilization of resources. However, MMH favors pipelines, since the blocks of different messages can be overlapped in the pipeline. The hardware architectures exploring the most efficient solutions for both single- and multi-message hashing applications are explained in subsequent sections. The *#MH* parameters for Keccak, Luffa and BMW are 5, 2 and 18, respectively.

5.1 Hardware Implementation of Keccak

One round of *KECCAK-f* permutation consists of the steps, θ, ρ, π, χ and $ï$. The top-level diagram of the hardware implementation for one round is given in Fig. 7. The architecture of the round is fully pipelined and operations in a pipeline stage are performed in a parallel fashion.

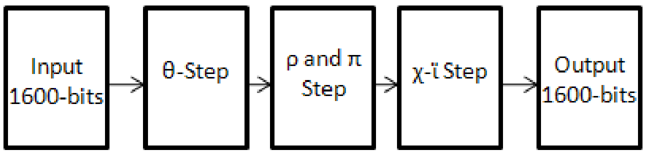


Fig. 7 High-Level Pipeline Stages of Keccak Hardware

Following the dependency graph of θ step, three-stage pipeline is used for its implementation, where each stage implements one sub-step of θ (*cf.* Fig. 1). Since new input arrives at each clock cycle in MMH, additional (synchronization) registers are required to forward the input to the later stages of θ step in addition to pipeline registers (*cf.* Fig. 8). θ step requires only bitwise *XOR* and cyclic shift operations by fixed amount. 50 bitwise *XOR* and 25 cyclic shift operations over 64-bit variables are performed in parallel in the hardware implementation of θ step.

The ρ and π step, implemented in one stage, utilizes cyclic shift operations where the shift amount depends on the position of the 64-bit element in the (5×5) state matrix. Since both operations are linear, instead of using two cyclic shifts, the hardware uses only one cyclic shift with a shift amount of ($r_\rho + r_\pi$). Only one pipeline stage is used to implement this operation. At each clock cycle the result of the cyclic shift operation is written to a register. It uses a total of 25 cyclic shift operations.

The final stage combines both χ and $ï$ steps. The χ operation is a combination of the *NOT*, *AND*, and *XOR* operations over 64-bit variables. Since $ï$ operation is only an *XOR* operation of 64-bit number at position of (0,0) of the 5×5 matrix it is done in the same pipeline stage with the χ operation. The cost of moving $ï$ to another pipeline step is 24×64 bit pipeline registers and since the operation is not in the critical path, we prefer doing it in the same pipeline stage with χ. The total number of 64-bit operations are 25 *NOT*, *AND*, and 26 *XOR*. The total 64-bit operations in one *KECCAK-f* permutation round is 76 *XOR*, 25 *NOT*, 25 *AND*, and 50 cyclic shift operations.

Increasing the pipeline stages, where it is unnecessary (i.e. partitioning non-critical paths), results in a greater area without increasing the operating frequency. In our design, we use an optimized pipeline partitioning as shown in Fig. 8 where the datapath with 5-stage pipeline is implemented for one round of *KECCAK-f* permutation.

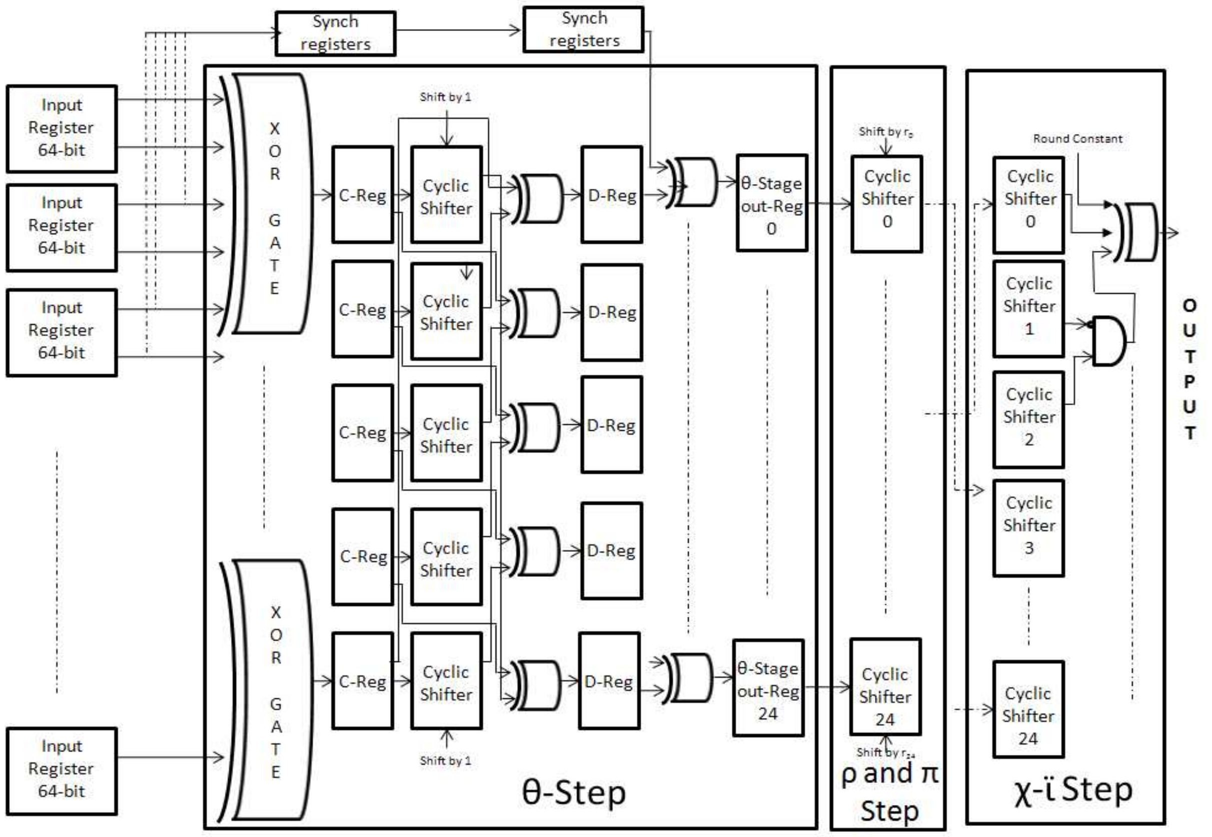


Fig. 8 Block Diagram of Keccak Hardware

For SMH implementation, no pipelining is used; i.e. one round is completed in one clock cycle, resulting in a lower operating frequency. All we do is to remove all the pipeline registers except for the output register to retain the output of one round of *KECCAK-f* permutation. Implementation results and the effects of design choices for both SMH and MMH architectures are discussed in Section 6.

5.2 Hardware Implementation of Luffa

Luffa Algorithm consists of two main modules as explained earlier; message injection (*MI*) and permutation (*P*), where *P* contains *Tweak* and *Step* modules. For each message block, *MI* and *Tweak* are performed only once while the *Step* modules Q_0, Q_1, and Q_2 are used for eight consecutive rounds as shown in Fig. 9.

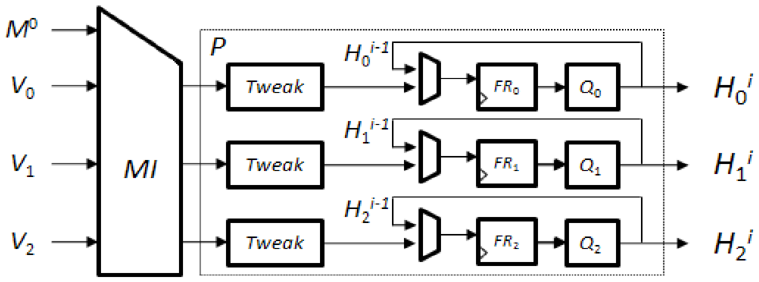


Fig. 9 Block Diagram of the Luffa Hardware Architecture

The block diagrams for *MI*, *Tweak* and *Step* modules are not given since each of them has a very regular flow resulting in straightforward architectures. ROMs are used to implement *S-Boxes* instead of multiplexers. Since the constants to be used in step functions are initially known, a module for constant generation is not implemented. The constant values of consecutive rounds are given sequentially.

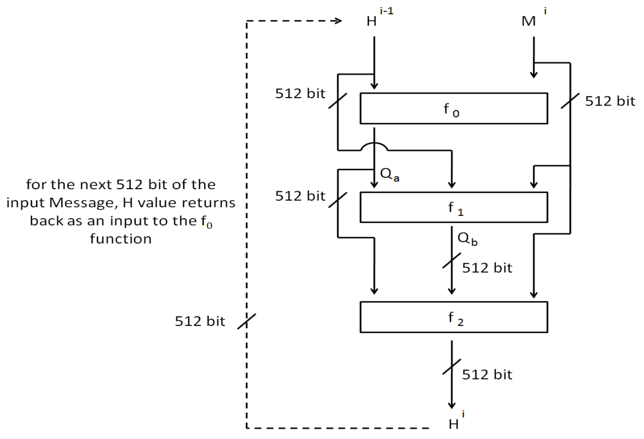
The critical path of message injection module consists of 3 *XOR* gates while the critical path of permutation module consists of 5 *XOR* gates and a read operation from ROM for *S-Boxes*. The cost of shift operations is negligible in hardware implementations since they are nothing more than interconnects.

Two different hardware designs are implemented for the Luffa algorithm. The initial design for SMH consists of a set of registers (FR_0, FR_1, FR_2 in Fig. 9) to forward the results of *Step* modules to the following round. Note that the registers are placed between the *Tweak* module and the *Step* modules instead of at the end of permutation module (*cf.* Fig. 9). This approach decreases the combination delay of a single round and increases the frequency noticeably at the cost of one extra cycle for message injection.

The second design is a high throughput architecture which is implemented to enhance the efficiency for MMH case. Since the Luffa algorithm has already a small combinational delay, we partition the *Step* function into two pipeline stages. The first stage consists of a ROM and two *XOR* gates while the second has 3 *XOR* gates. This division is done by inserting the pipeline stage after the second round of the Feistel ladder in *MixWord* functions. Addition of a single register stage increases the frequency of hardware for about 35% at the cost of 8% area overhead for ASIC implementation as shown in Table 2.

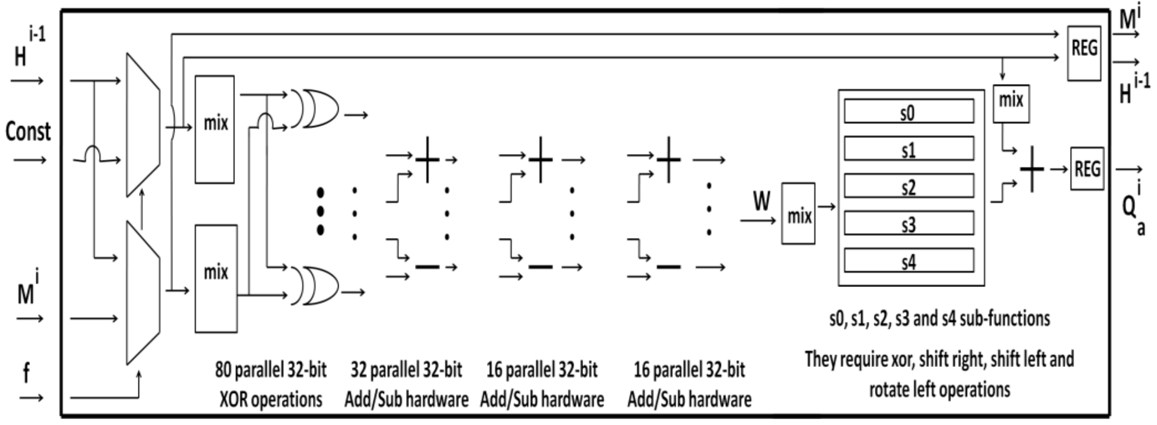
5.3 Blue Midnight Wish Hardware Implementation

Top level block diagram of the proposed pipelined and parallel hardware architecture for the implementation of BMW algorithm is shown in Fig. 10. BMW algorithm does not use multiple rounds for hashing of a single message block. As shown with dashed line in Fig. 10, the resulting value H^i is used as input in the processing of the next message block. Hardware architecture of f_0 for MMH is shown in Fig. 11, where the computation starts with mixing the bits of the previous hash block (H^{i-1}) and message block (M^i). Mixing can be implemented by only wiring in hardware. Since the operations are not in the critical path, f_0 is implemented as a single pipeline stage. In SMH version of the architecture, the registers at the output (*cf.* Fig. 11) are removed.



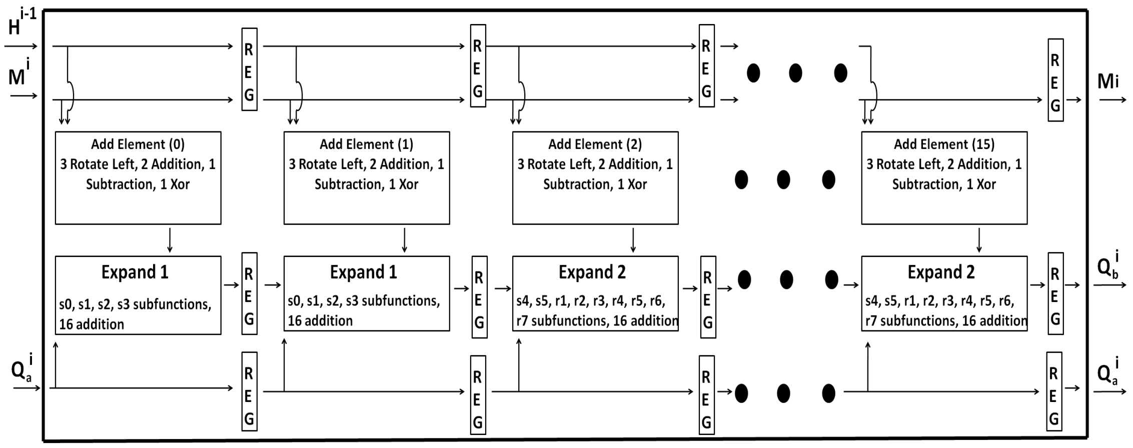
H^{i-1}
M^i
512 bit
f_0
512 bit
for the next 512 bit of the input Message, H value returns back as an input to the f_0 function
Q_a
512 bit
f_1
Q_b
512 bit
512 bit
f_2
512 bit
H^i

Fig. 10 Block Diagram of the BMW Hardware Architecture



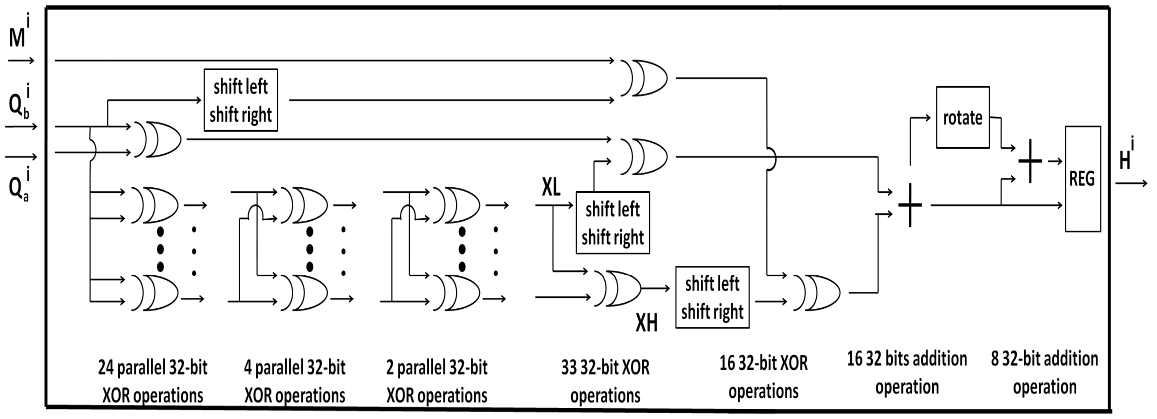
H i-1
Const
M i
f
mix
mix
W
mix
mix
s0
s1
s2
s3
s4
REG
REG
M i
H i-1
Q i a
80 parallel 32-bit XOR operations
32 parallel 32-bit Add/Sub hardware
16 parallel 32-bit Add/Sub hardware
16 parallel 32-bit Add/Sub hardware
s0, s1, s2, s3 and s4 sub-functions
They require xor, shift right, shift left and rotate left operations

Fig. 11 Hardware Architecture of f_0 function



H i-1
M i
Q i a
Add Element (0) 3 Rotate Left, 2 Addition, 1 Subtraction, 1 Xor
Add Element (1) 3 Rotate Left, 2 Addition, 1 Subtraction, 1 Xor
Add Element (2) 3 Rotate Left, 2 Addition, 1 Subtraction, 1 Xor
Add Element (15) 3 Rotate Left, 2 Addition, 1 Subtraction, 1 Xor
Expand 1 s0, s1, s2, s3 subfunctions, 16 addition
Expand 1 s0, s1, s2, s3 subfunctions, 16 addition
Expand 2 s4, s5, r1, r2, r3, r4, r5, r6, r7 subfunctions, 16 addition
Expand 2 s4, s5, r1, r2, r3, r4, r5, r6, r7 subfunctions, 16 addition
REG
Mi
Q i b
Q i a

Fig. 12 Hardware Architecture of Pipelined f_1 function



M i
Q i b
Q i a
shift left shift right
XL
shift left shift right
XH
shift left shift right
rotate
REG
H i
24 parallel 32-bit XOR operations
4 parallel 32-bit XOR operations
2 parallel 32-bit XOR operations
33 32-bit XOR operations
16 32-bit XOR operations
16 32 bits addition operation
8 32-bit addition operation

Fig. 13 Hardware Architecture of f_2 function

Hardware architecture of the function f_1 for MMH is shown in Fig. 12. f_1 requires two *expand*1 and 14 *expand*2 sub-functions, each of which uses *s* and *r* sub-functions, 16 additions, and *AddElement*[2] operations. Since each *expand* function waits for the previous *expand* function to complete, they cannot work in parallel. This necessitates registering of 512-bit H^{i-1}, M^i and Q^i values to forward them throughout the pipeline. For MMH version, 16 pipeline stages are used to implement f_1 while for SMH hardware, all pipeline and synchronization registers are removed.

[2] See the original description of BMW algorithm for AddElement operation [4].

Hardware architecture of f_2 function implementation for MMH is shown in Fig. 13. f_2 takes 512-bit M^i, Q_a^i and Q_b^i as inputs, and generates 512-bit H^i. It basically compresses 2048-bit input to the 512 bits. In the finalization step, the leftmost 256 bits of the H^{final} forms the hash of the message. f_2 is implemented as a single pipeline stage and it is the same for both SMH and MMH.

6. Implementation Results

We implement our designs using Verilog HDL, and the hardware implementation results of the three candidate hash algorithms, namely Keccak, Luffa and BMW are given in Table 2, where SMH and MMH stands for architectures optimized for single- and multi-message hashing, respectively. The table contains the frequency, latency, area, throughput, and efficiency results for target FPGAs and ASIC synthesis. *Efficiency*[3] in the last column is a metric defined as the throughput per unit of resources. We provide *Efficiency* metric mainly for comparing *our* hardware implementations of the candidate algorithms. Efficiency metric should be carefully used for realizations that use the same (or close) target technology and the tool chain. Xilinx XST is used for FPGA synthesis and Synopsys Design Compiler with Synopsys 90nm Generic Library under typical operating conditions is used for ASIC synthesis. The area is given in terms of gate equivalent (GE) and slice count. The hardware results are also compared with the implementation results of the designs in [6][7][8][11][12].

One natural result that can be observed from Table 2 for all designs is that total area is lower in SMH implementations. This is due to the fact the SMH architectures do not have pipeline and synchronization registers. Pipelined datapath in MMH, on the other hand, significantly increases the throughput due to the increase in the operating frequency at the cost of additional area. In what follows, we summarize, evaluate, and interpret implementation results for each algorithm.

Keccak: SMH variant of Keccak provides high throughput, comparable to Luffa, while it has the lowest resource (slice or GE) usage. One of its advantages over Luffa and BMW algorithms to achieve high throughput is its large *BlockSize*, which is 1088 bits, whereas the *BlockSize* of Luffa and BMW is 256 and 512 bits, respectively. Its *efficiency* is again very high and almost the same as Luffa's. To the best of our knowledge, MMH implementation of Keccak is the first in the literature and we observe that the algorithm is very suitable for MMH applications. It is again relatively low area and high throughput architecture which provides the highest efficiency for all target devices but Virtex 2, where its efficiency is very close to the best (i.e., BMW). When compared with Luffa, its efficiency improves much better for MMH. One notable fact is that its MMH variants can achieve the highest clock frequency in MMH except for Virtex 2.

Luffa: SMH variant has throughput values comparable to (in most cases higher than) Keccak. Except for ASIC, it has the highest throughput among the three. It is relatively low area architecture. Our SMH variant is superior to the architecture in [11], implemented using a comparable ASIC technology, in terms of area and efficiency. Luffa, however, does not gain speedup from pipelining (for MMH) as much as Keccak and BMW do. It takes, on the other hand, very little extra area to obtain MMH variant, which consumes the lowest area in each target technology. When our MMH variant of Luffa is compared against the only MMH implementation in literature [8], one can observe that ours provides a much better alternative thanks to its high efficiency metric (*cf.* 2.83 and 0.74). Note that the difference in efficiency values will not change much even if the technology differences are accounted for. Having much higher efficiency enables replication of MMH architecture to achieve even higher throughput values. For example, replicating our MMH variant of Luffa 10 times will result in a throughput of 350 Gbit/s. This justifies the area efficient design approach for high speed MMH applications.

BMW: In every case, BMW consumes much more area than the other algorithms due to the fact that it does not use round function approach whereby a simple function is repeatedly executed many times. The SMH variant is the slowest for FPGA realizations and it cannot achieve high clock frequency values. A surprising result is that our implementation of SMH variant is the fastest in ASIC realization; a result, which is similar to [11]. One reason for its poor performance in FPGA realizations is related to the fact that it does not use round function approach. Simple, but many, arithmetic/logical operations employed in BMW in a long datapath, if implemented using LUTs, will tend to incur high area and long interconnect delay while ASIC realizations will not suffer from the same problem. However, BMW, due to its high area consumption, may suffer from its poor efficiency values in each case for the applications where both speed and area constraints are important. Due to its structure, BMW is suitable for pipelining since its high latency datapath can be partitioned evenly and profitably. Therefore, its MMH variant

[3] Efficiency values of FPGAs are not comparable to efficiency values of ASIC realizations.

Table 2. The hardware implementation results of SHA-3 candidates.

	Algorithm	Hashing Method	Target Technology	Frequency (MHz)	# of Rounds	Latency (cycles)	Area (Slices/GE)	Throughput (Gbits/s)	Efficiency = Throughput/ (Area×10^6)
our hardware implementation results	Keccak	SMH	Spartan 3	81.4	24	25	2,024	3.46	1.71
	Keccak	MMH		338.3	24	121	4,356	14.85	3.41
	Luffa	SMH		157.3	8	9	2,956	4.37	1.48
	Luffa	MMH		202.9	8	17	3,733	5.97	1.60
	BMW	SMH		4.22	1	1	10,531	2.11	0.20
	BMW	MMH		57.4	1	18	12,477	28.7	2.30
	Keccak	SMH	Virtex 2	136.6	24	25	2,024	5.81	2.87
	Keccak	MMH		341.5	24	121	4,356	14.99	3.44
	Luffa	SMH		301.4	8	9	2,952	8.37	2.84
	Luffa	MMH		424.7	8	17	3,721	12.49	3.36
	BMW	SMH		6.71	1	1	10,432	3.36	0.32
	BMW	MMH		86.3	1	18	12,244	43.15	3.52
	Keccak	SMH	Virtex 4	142.9	24	25	2,024	6.07	3.00
	Keccak	MMH		508.7	24	121	4,356	22.33	5.13
	Luffa	SMH		308.2	8	9	2,989	8.56	2.86
	Luffa	MMH		470.8	8	17	3,719	13.85	3.72
	BMW	SMH		9.01	1	1	10,486	4.51	0.43
	BMW	MMH		115.96	1	18	12,497	57.98	4.64
	Keccak	SMH	90nm ASIC	454.5	24	25	10.5k	19.32	1.84
	Keccak	MMH		1,694.9	24	121	23.2k	74.41	3.21
	Luffa	SMH		769.2	8	9	11.5k	21.37	1.86
	Luffa	MMH		1,204.8	8	17	12.5k	35.44	2.83
	BMW	SMH		52.63	1	1	55.9k	26.32	0.47
	BMW	MMH		265.96	1	18	160.1k	132.98	0.83
[12]	Keccak	SMH	Spartan 3A	85	24	25	3393	4.8	1.41
	Keccak	SMH	Virtex 5	122	24	25	1412	6.9	3.61
[6]	Luffa	SMH	Altera S-3	47.04	1	1	16,552	12.042	0.73
	BMW	SMH	Altera S-3	9.55	1	1	12,917	4.889	0.38
[7]	Keccak	SMH	180nm ASIC	487.80	24	25	56.31k	21.23	0.38
	Luffa	SMH	180nm ASIC	483.09	8	9	44.9k	13.74	0.31
	BMW	SMH	180nm ASIC	10.46	1	1	169k	5.358	0.03
[8]	Luffa	SMH	130nm ASIC	1124	8	9	30.8k	31.9	1.07
	Luffa	MMH	130nm ASIC	508	1	9	156.6k	115.5	0.74
[11]	BMW	SMH	90nm ASIC	52.08	1	1	60.0k	26.66	0.54
	Luffa	SMH	90nm ASIC	100.4	1	1	68.9k	25.70	0.70

provides the highest throughput (132.98 Gbits/s) among all designs due to the fact that deep pipelining improves clock frequency up to five times. In this respect, BMW benefits from the pipelining much more than Luffa and Keccak do. Another important observation is that FPGA realizations do not observe a significant area increase between SMH and MMH (less than 20%) as in the case of ASIC where increase is almost three fold. The relatively lesser increase of area in FPGAs can be attributed to the better utilization of registers in FPGA slices in MMH architecture, which are usually wasted in SMH case. The MMH variant has the worst efficiency compared to the MMH variants of Luffa and Keccak.

Note that the numbers of pipeline stages in Keccak and Luffa implementations, which are directly related to *#MH* and throughput, are significantly lower than that of BMW. The number of pipeline stages can also be increased (therefore throughput, too) if their rounds are un-rolled. However, as can be deducted from Eq. 2, further increase in the number of pipeline stages *in this manner* will increase throughput and area roughly at the same rate, which in turn cannot increase the efficiency significantly. In other words, hardware replication and round unrolling result in a similar effect for MMH applications. In this respect, our choices for the numbers of pipeline stages for each design provide optimal configurations for efficient and high throughput hash computations.

7. Conclusion

We presented efficient, high throughput hardware implementations of SHA-3 candidates Keccak, Luffa and Blue Midnight Wish with an emphasis on MMH applications. We basically employed pipelining, parallelism and re-timing techniques to improve the performances and efficiencies of our designs. Implementation results of six different architectures are provided for ASIC and three different FPGA devices. We compared our results with those in literature when possible and fair. To the best of our knowledge, this is the first work that provides a comparative analysis of three high performance SHA-3 candidates for MMH applications. We also evaluated and commented on the results and give a perspective for each candidate to increase the insight on different aspect of the hardware performance of each. Our architectures provide the highest efficiency values in majority of the cases.

References

[1] W. Diffie, M. E. Hellman: New directions in cryptography. IEEE Trans. On Information Theory IT-22 (6), 644-654 (1976).

[2] M. O. Rabin: Digitalized signatures. In: Lipton, R., DeMillo, R. (eds.) Foundations of Secure Computation, pp. 155–166. Academic Press, NY (1978).

[3] R. Merkle: Secrecy, Authentication, and Public Key Systems. UMI Research Press (1979).

[4] National Institute of Standards and Technology (NIST). Cryptographic Hash Algorithm Competition Website. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[5] H. Bar-El, Security Implications of Hardware vs. Software Cryptographic Modules. Discretix White Paper. October 2002.

[6] ECRYPT II. SHA-3 Hardware Implementations. http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations.

[7] S. Tillich et al: High-Speed Hardware Implementations of BLAKE, Blue Midnight, Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD and Skein. Cryptology ePrint Archive. October 2009.

[8] M. Knezevic, I. Verbauwhede: Hardware Evaluation of the Luffa Hash Family. *4th Workshop on Embedded Systems Security 2009*, Grenoble, France.

[9] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, “Sponge Functions,” Ecrypt Hash Workshop 2007.

[10] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, “On the Indifferentiability of the Sponge Construction,” Advances in Cryptology, Eurocrypt’08, Lecture Notes in Computer Science, Vol. 4965, Springer-Verlag, pp. 181–197, 2008.

[11] A. H. Namin and M. A. Hasan, “Hardware Implementation of the Compression Function for Selected SHA-3 Candidates”.

[12] J. Strömbergson, Implementation of the Keccak Hash Function in FPGA Devices, 2009.

[13] X. Wang, Y. L. Yin, H. Yu: Finding Collisions in the Full SHA-1. CRYPTO 2005: 17-36.

[14] K. Aoki, J. Guo, K. Matusiewicz, Y. Sasaki, L. Wang: Preimages for Step-Reduced SHA-2. ASIACRYPT 2009: 578-597.